CS 2223 - Final Project
2-Dimensional KD Trees

Andrew Fisher
Denver Blake
Uri Dvir
John Max Petrarca
Ivan Martinovic

Introduction

A KD Tree is a spatial tree structure used to efficiently perform nearest neighbor queries in k dimensions. For each point *A* which resides in our k-dimensional space and which is inserted into the KDTree, a node in the KD Tree contains the coordinates of the point, the region it partitions and a coordinate axis which determines the orientation of the partitioning. The root node partitions the entire k-dimensional space along the first coordinate axis x_1.

For each point *A* the partitioning is performed by cutting the k-dimensional region with a *(k-1)* dimensional object passing through *A* which is normal to the orientation of the partitioning. Suppose our k-dimensional space is defined with k coordinate axes: $x_1, x_2 \ldots x_k$. The partitioning axis alternates, such that if the axis for the parent is x_n the axis for the child will be x_{n+1}. If the axis for the parent happens to be x_k the axis for the child will be x_1.

Each node contains two children *smaller* and *bigger* referring to the child having a smaller or larger coordinate corresponding to the partitioning axis of the parent. While inserting nodes into a KD tree, a new node N_c will be a child to another node N_p if it lies within the region which N_p partitions and if N_p doesn't already have the appropriate *smaller* or *bigger* child.

If the orientation of the parent's region's partitioning was the axis x_n then the region which the *smaller* child now partitions will be bounded by the region which its parent was partitioning from its lower bound along the x_n axis all the way up to the parent's x_n coordinate. Conversely the region which the *bigger* child now partitions will be bounded by the region which its parent was partitioning from the parent's coordinate along the x_n axis all the way up to the region's upper bound along x_n

The KD tree spacial structure was first invented in 1976 by Jon Louis Bentley and Michael Ian Shamos in their thesis: "Divide-and-Conquer in Multidimensional Space".[1]

[1] Bentley, J. L. & Shamos, M. I., *"Divide-and-Conquer in Multidimensional Space"*, Carnegie-Mellon University, published 1976

2-Dimensional KD Tree

The main objective of our project was to implement a 2 Dimensional KD Tree. Such a tree would be used to divide a two-dimensional coordinate plane using horizontal and vertical lines parallel to the coordinate axes for better facilitating nearest neighbor search queries. For better visualization of how 2D space is being partitioned, consider the **Figure 1** showing how points 0 through 5 were added to the KD tree:

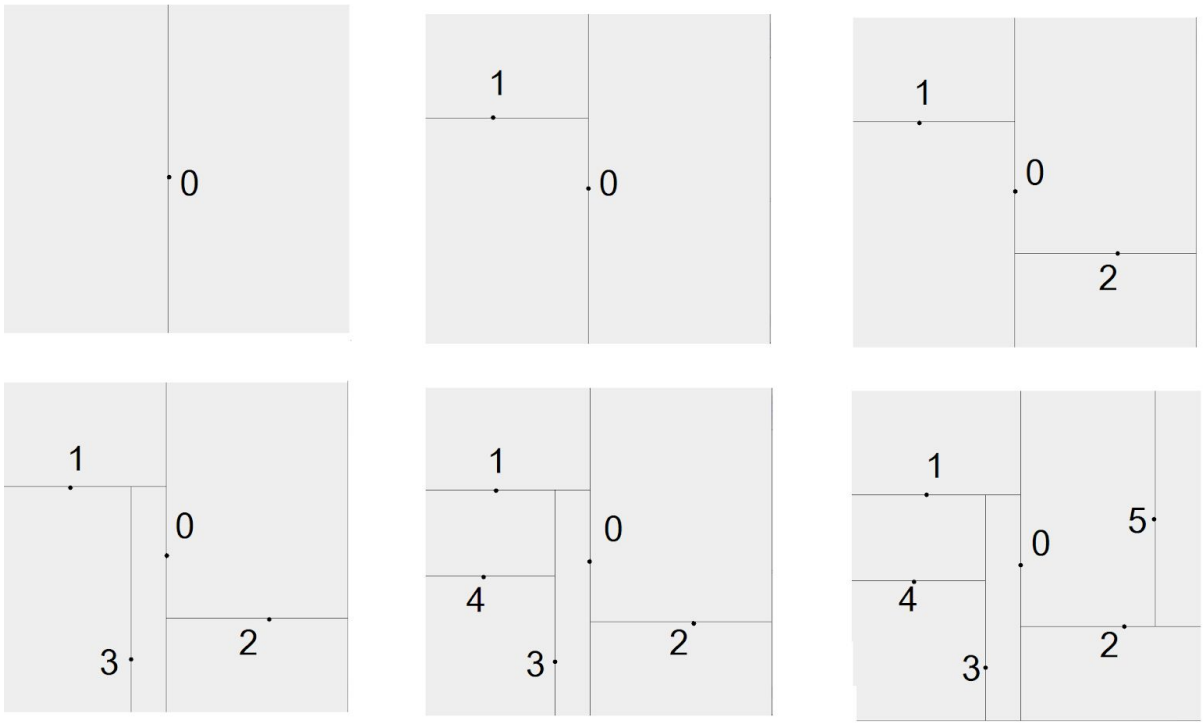

Figure 1. Sequence of points added to a 2 Dimensional KD Tree

Initially we have an empty 2D plane in which we add point 0. We initially partition with respect to the x-axis i.e. the line drawn is vertical through 0.

Then we add point 1 to the left of 0. Since 1 is in the subregion of 0, its orientation of partitioning is now opposite to the orientation of partitioning of 1 i.e. the line through 1 is horizontal.

Similarly to the situation of point 1, point 2 is now added to the right subregion of 0 and the line through 2 is horizontal.

Point 3 is then added and lies in the subregion of 1. Therefore the orientation of its partitioning is again flipped but this time with respect to 1. Hence the line drawn through 3 is vertical.

Similarly 4 is added in the subregion of 3 and the line through 4 is horizontal.

Finally point 5 is added in the subregion of point 2. Since line through point 2 is horizontal, the line through point 5 will be vertical.

Worst-case Scenario for Adding Elements to KD Trees

The code for adding an element to the KD Tree is similar to the code for adding an element to a Binary Search tree.

```
public KDNode add(KDNode parentNode, double x, double y) {
    boolean bigger = biggerThanNode(parentNode,x,y);
    Region newRegion = parentNode.getSubRegion(bigger);
    KDNode nodeAdded;
    //base case checks if a new node is to be added
    //it checks whether the bigger and smaller subTrees are null and checks if the corresponding
    //coordinate justifies adding the new point as the bigger or smaller subtree

    if (bigger && parentNode.bigger == null) {
        nodeAdded = new KDNode(x,y, flipOrientation(parentNode.orientation), newRegion);
        parentNode.bigger = nodeAdded;
        return nodeAdded;
    }
    else if (!bigger && parentNode.smaller == null) {
        nodeAdded = new KDNode(x,y, flipOrientation(parentNode.orientation), newRegion);
        parentNode.smaller = nodeAdded;
        return nodeAdded;
    }
    // now to the recursive case:
    // we just have to see to which on which subtree to call the recursion on
    if (bigger) {
        return add(parentNode.bigger, x, y);
    }
    else {
        return add(parentNode.smaller,x, y);
    }
}
```

Figure 2. Java implementation of the recursive add() function for the 2 Dimensional KD Tree

The base case is when the parent respectively doesn't have a bigger or smaller child and the newly added KDNode is either bigger or smaller with respect to the partitioning axis of the parent . If it was determined that the base case is not satisfied, then it must be determined on which one of the parents children the recursive call is going to be made.
The biggerThanNode method takes as input a node and a point's coordinates x and y, then returns true if the coordinate x or y is bigger than the corresponding coordinate of the node, based on the node's orientation of partition.

Looking at the code we can conclude that the sequence of points which produce the worst case scenario for adding is the one with the most amount of recursive calls. This occurs if the newly added point is in the subregion of the point which was previously added. Although there are infinitely many patterns which produce this worst-case behaviour, two common ones are zig-zag patterns and spiral patterns (**Figure 3**).

Figure 3. The zig-zag pattern (left) and the spiral pattern (right) which create the worst performance for the add() function

In such a case when adding the N-th element the number of comparisons i.e. calls to biggerThanNode function is (N-1) and number of calls to add() is also(N-1).Therefore the total amount of recursive calls and comparison calls to add N elements in total is N*(N-1)/2. Comparison of the average random case and worst case for adding N elements can be seen below on **Diagram 1**

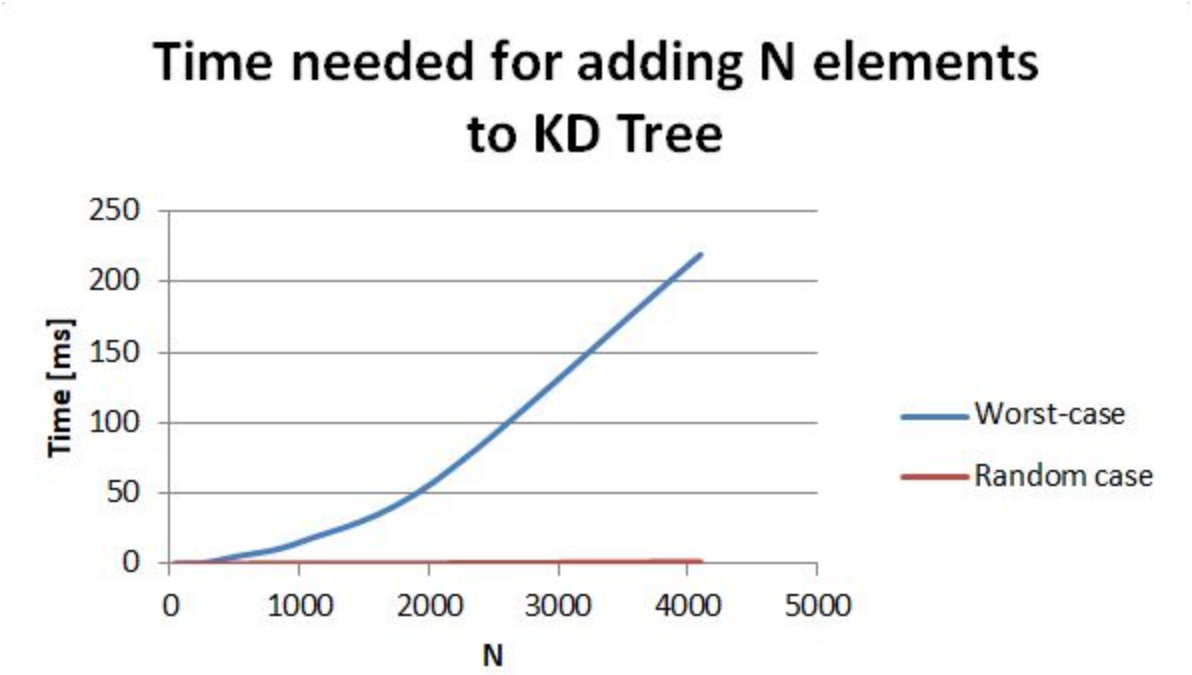

Diagram 1 Worst-case (blue) vs Random case (red) when adding elements to the KD Tree

For reference, average case barely exceeds 1 ms, hence the worst case scenario is on average in this small sample 45 times slower.

Evaluation of Nearest Neighbor Performance

Table 1 contains information on the average time and average number of distance calculations when Nearest-neighbor query was ran 1000 times on 100 different KD trees with N elements:

N	time (ms)	GetDistance calls
32	0.000252	41.80674
64	0.000266	50.83393
128	0.000319	60.01835
256	0.000376	67.73414
512	0.000443	75.70471
1024	0.000522	83.66873
2048	0.0006	90.36106
4096	0.000713	97.01396
8192	0.000891	104.57047
16384	0.001191	110.88089
32768	0.001588	117.53313
65536	0.002112	124.91409
131072	0.002565	130.90007
262144	0.002899	137.8982

Table 1 Table containing the number of points in a KD Tree and the corresponding values for the average time it takes to perform a nearest neighbor search query as well as the average number of distance calculations needed for performing the same search queries.

The time curve and its best-fit lines are presented in **Diagram 2**

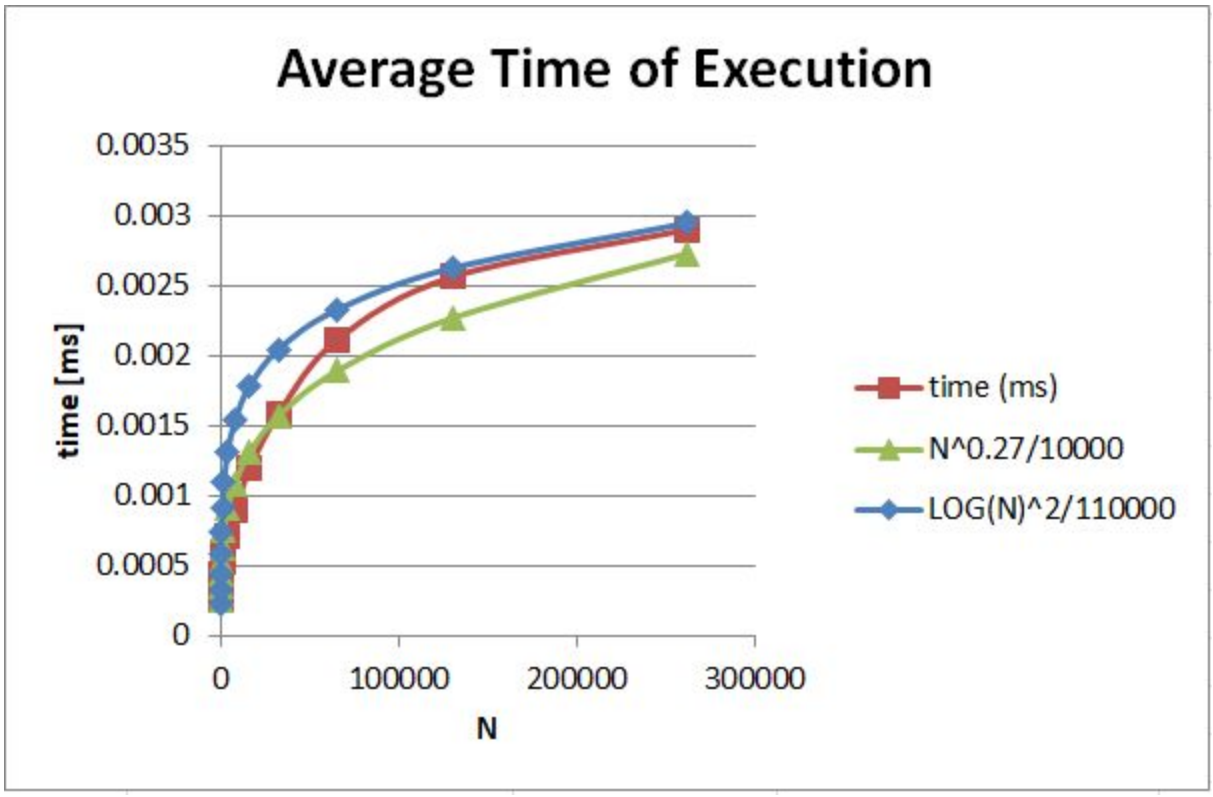

Diagram 2 Average time for finding nearest neighbor(red) along the logarithmic (blue) and power (green) best fit lines when performing a nearest neighbor search query

For the average time of execution it was found that the performance is somewhere between the order N^0.27 and (log(N))^2. The logarithmic function seems to initially grow faster than the actual function, whereas the power function initially grows at the

same pace. In the end the logarithmic function seems to start growing slower than the actual function, whereas the power function starts growing faster.
The average number of distance calculations seems to be simpler to approximate, as presented in **Diagram 3**.

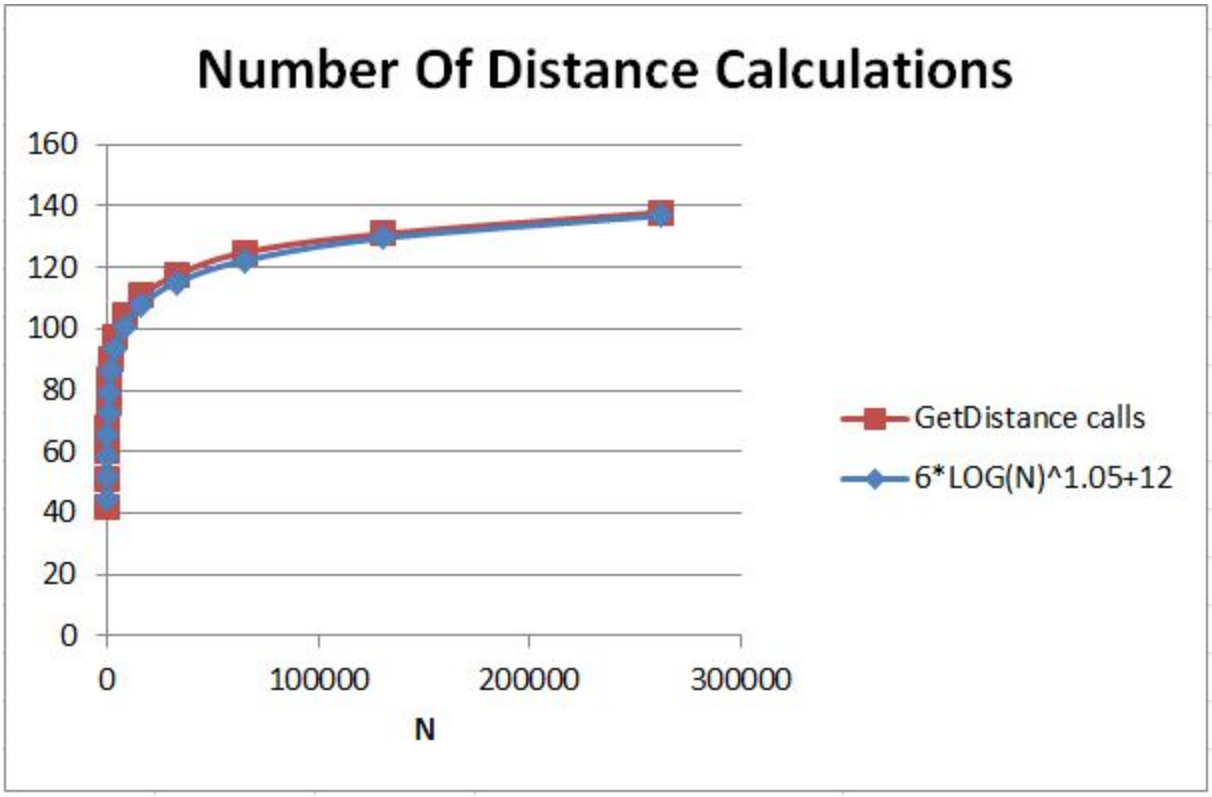

Diagram 3 Average number of distance calculation (red) and the logarithmic best fit line (blue) when performing a nearest neighbor search query

The order of growth for the average number of distance calculations appears to be (log(N))^1.05.

Nearest Neighbor query: KD Tree vs Brute Force Algorithm

Table 2 was constructed from the average time of execution of a brute force algorithm and our KD Tree algorithm. From the table we may see that our KD Tree Algorithm starts outperforming a brute force algorithm for a set of points containing somewhere between 128 and 256 points.

N	KD Algorithm Time [ms]	Brute Force Algorithm Time [ms]
32	0.000252	0.000164
64	0.000266	0.000195
128	0.000319	0.000288
256	0.000376	0.000477
512	0.000443	0.000817
1024	0.000522	0.001706
2048	0.0006	0.003271
4096	0.000713	0.006362
8192	0.000891	0.012877
16384	0.001191	0.025486
32768	0.001588	0.051634
65536	0.002112	0.121298
131072	0.002565	0.252744
262144	0.002899	0.588444

Table 2 Table containing the number of points in a set N, with corresponding average time needed to perform a KD Tree nearest neighbor search query and a brute force nearest neighbor query

Conclusion

KD Trees present an interesting approach for partitioning data in multi-dimensional space. They were one of the first efficient data structures which allowed multidimensional data analysis and geometric optimization. They were also the very first efficient data structure used for computing nearest and farthest neighbor queries in multi-dimensional space. [2]

The boost in performance they provide when compared to brute force algorithms is tremendous in the context of performing nearest neighbor queries. However just like regular binary search trees they can be extremely inefficient when unbalanced, both when adding elements and performing data analysis.. These cases can occur when data has a sort of multi-dimensional zig-zag or spiral pattern.

Nevertheless with a self balancing system, their performance for performing nearest neighbor and farthest neighbor search queries is optimal. [3]

[2] Bentley, J. L. & Shamos, M. I., *"Divide-and-Conquer in Multidimensional Space"*, Carnegie-Mellon University, published 1976

[3] Bentley, J. L. & Shamos, M. I., *"Divide-and-Conquer in Multidimensional Space"*, Carnegie-Mellon University, published 1976